Numerical calculations using pySecDec

Vitaly Magerya

Institute for Theoretical Physics,
Karlsruhe Institute of Technology

Precision calculations for future e^+e^- colliders,
16 June, CERN

Where does pySecDec fit in?

Basic steps of calculating scattering matrix elements:

1. Generate Feynman diagrams for the process.
   - $\mathscr{M}$ = + + + ...
2. Project the diagrams onto scalar integrals.
   - $\mathscr{M} = C_1 + C_2 + C_3 + C_4 + \ldots$
3. Reduce the number of integrals using IBP relations.
   - $\mathscr{M} = C_1' + C_2' + C_3' + \ldots$
4. *Evaluate the loop integrals.*
   - Analytically: hard to impossible at two loops if masses are present.
   - *Numerically:*
     - *Sector decomposition:* pySecDec, Fiesta, sector_decomposition.
     - Mellin-Barnes representation: MB, Ambre.
     - Numerical differential equations: DiffExp, AMFlow, etc.
5. Integrate over kinematics.
   - A whole field of study by itself.

pySecDec overview

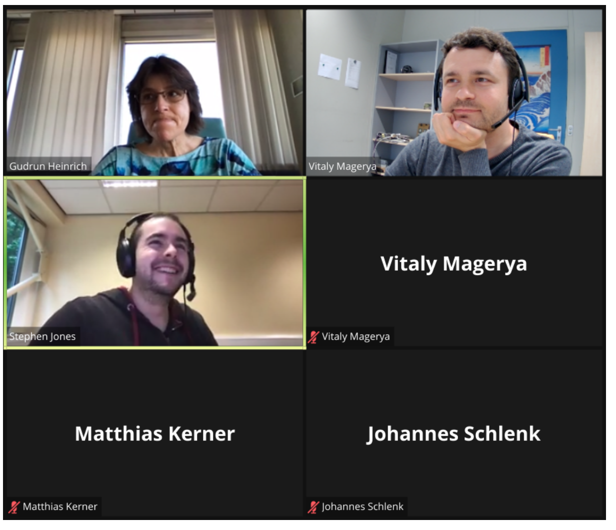
pySecDec: library for *numerically evaluating parametric integrals* via sector decomposition and Monte Carlo integration. [Heinrich et al '21, '18, '17]

- https://github.com/gudrunhe/secdec
- Written in *Python*, C++, Form.
- Multiple sector decomposition methods: iterative, geometric.
- Multiple integration algorithms:
  - Best: Randomized Quasi Monte Carlo (Qmc); [Borowka et al '18]
  - Classic: Vegas/Suave/Divonne/Cuhre (Cuba), Cquad (Gsl).
- CPU & GPU support.

Recent new features (release v1.5):

- Quality-of-life improvements:
  - Installation via the *standard Python package* installer (Linux, Mac):
    `pip3 install --user pySecDec`
  - Adaptive contour deformation (no more "sign check errors"), and Form configuration (no more "insufficient WorkSpace" errors).
- Adaptive sampling of whole amplitudes (weighted sums of integrals).
- Builtin asymptotic expansion of integrals.

The pySecDec team



Not pictured: Sophia Borowka, Stephan Jahn, Florian Langer, Andres Poldaru, Emilio Villa, Tom Zirke.

Sector decomposition in short

$$I = \int_0^1 \mathrm{d}x \int_0^1 \mathrm{d}y \, \left(x + y\right)^{-2+\varepsilon} = ?$$

Problem: the integrand diverges at $x, y \to 0$, can't integrate numerically.

Solution: [Heinrich '08; Binoth, Heinrich '00]

1. Factorize the divergence in x and y with sector decomposition:
   - $$I = \int \cdots \times \Big(\underbrace{\theta\left(x > y\right)}_{\text{Sector 1}} + \underbrace{\theta\left(y > x\right)}_{\text{Sector 2}}\Big) = \int_0^1 \mathrm{d}x \int_0^x \mathrm{d}y \, \left(x + y\right)^{-2+\varepsilon} + \begin{pmatrix} x \\ \updownarrow \\ y \end{pmatrix}$$
2. Rescale the integration region in each sector back to a hypercube:
   - $$I \overset{y \to xy}{=} \int_0^1 \mathrm{d}x \, \underbrace{x^{-1+\varepsilon}}_{\text{Factorized pole}} \int_0^1 \mathrm{d}y \left(1 + y\right)^{-2+\varepsilon} + \begin{pmatrix} x \\ \updownarrow \\ y \end{pmatrix}$$
3. Extract the pole at $x \to 0$ analytically, expand in ε:
   - $$I = -\frac{2}{\varepsilon} \int_0^1 \mathrm{d}y \left(1 + y\right)^{-2+\varepsilon} = -\frac{2}{\varepsilon} \int_0^1 \mathrm{d}y \left(\frac{1}{\left(1+y\right)^2} - \frac{\ln\left(1+y\right)}{\left(1+y\right)^2} \varepsilon + \mathscr{O}\left(\varepsilon^2\right) \right)$$
4. Integrate each term in ε numerically (they all converge now).

Contour deformation

$$I \equiv \int \mathrm{d}^n\vec{x}\, \frac{U^\alpha\left(\vec{x}\right)}{F^\beta\left(\vec{x}, \ldots\right) + i0}$$

What to do if $F\left(\vec{x}\right) = 0$ inside the integration region? Deform $\vec{x}$ into the complex plane:

$$\vec{x} \longrightarrow \vec{x} + i\,\vec{\delta}\left(\vec{x}\right),$$

$$F \longrightarrow F + i\,\delta\,\partial F - \delta^2\,\partial^2 F - i\,\delta^3\,\partial^3 F + \ldots\,.$$

Choose $\vec{\delta}\left(\vec{x}\right)$ to enforce the $+i0$ prescription:

$$\operatorname{Im} F = \delta\,\partial F - \delta^3\,\partial^3 F + \ldots \overset{\delta\to 0}{\approx} \delta\,\partial F > 0 \quad \Rightarrow \quad \vec{\delta}\left(\vec{x}\right) = \lambda\,\vec{\partial} F\left(\vec{x}\right).$$

- There is always λ small enough that $\delta\,\partial F > \delta^3\,\partial^3 F$, and $\operatorname{Im} F > 0$.
- The larger λ is, the further the pole is, the better the convergence is.
- In practice: choose λ heuristically, but decrease it if $\operatorname{Im} F\left(\vec{x}_i\right) < 0$.
  - Gradient-based λ optimization can be useful. [Winterhalder et al '21]

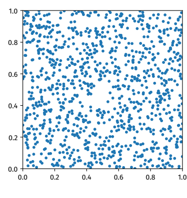
Monte Carlo Integration

To numerically approximate an n-dimensional integral

$$I \equiv \int_0^1 \mathrm{d}^n\vec{x}\, I(\vec{x})\,,$$

Monte Carlo integration:

1. Sample $\vec{x}_{1\ldots N}$ uniformly from $[0;1)^n$.
2. Estimate I as $\mathrm{mean}\big(I(\vec{x}_i)\big)$.
3. Estimation error is $\sqrt{\frac{1}{N}\,\mathrm{stdev}\big(I(\vec{x}_i)\big)} \sim N^{-\frac{1}{2}}$.



In pySecDec: Vegas/Suave/Divonne integrators via Cuba. [Hahn '04]

Monte Carlo, Quasi

Quasi Monte-Carlo integration:

1. Let $\vec{x}_{1\ldots N}$ be a *low-discrepancy sequence* in $[0;1)^n$.
2. Estimate I as $\mathrm{mean}\big(I(\vec{x}_i)\big)$.
3. Error is $V(I)\,D(\vec{x}_i)$, where
   - V is the *Hardy-Krause variation* of I (impractical to calculate),
   - and D is the *star discrepancy* of $\{\vec{x}_1, \ldots, \vec{x}_N\}$,

$$D(\vec{x}_i) \equiv \max_{Q \subset [0;1)^n} \left| \frac{\text{number of } \vec{x}_i \in Q}{N} - \text{volume of } Q \right|.$$

Low-discrepancy sequences: [Dick, Kuo, Sloan '13]

- digital sequences (Sobol', Faure, etc), with error $\sim N^{-1}$;
- Owen's scrambled nets, with error $\sim N^{-1.5}$ if $\partial_x I$ is square-integrable;
- lattice rules, with error $\sim N^{-\alpha}$, if $\partial_x^{(\alpha)} I$ is square-integrable and periodic.
  - To enforce periodicity, use a transformation ($x \to y$):
    - baker's, $y(x) = 1 - |2x - 1|$;
    - Korobov, $\mathrm{d}y/\mathrm{d}x \sim x^l\,(1-x)^r$; etc.

Monte Carlo, Randomized Quasi

Randomized Quasi Monte-Carlo integration:

1. Let $\vec{x}^{(k)}_{1\dots N}$ be K random low-discrepancy sequences in $[0;1)^n$.
2. Estimate I as $\mathrm{mean}_k\Big(\mathrm{mean}_i\Big(I\Big(\vec{x}^{(k)}_i\Big)\Big)\Big)$.
3. Error is $\sqrt{\frac{1}{K}\,\mathrm{stdev}_k\Big(\mathrm{mean}_i\Big(I\Big(\vec{x}^{(k)}_i\Big)\Big)\Big)}$.

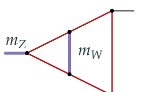
In pySecDec: the Qmc integrator, using rank-1 lattice rules: [Borowka et al '18]

$$\vec{x}^{(k)}_i = \left(i\,\vec{g}_N + \vec{\Delta}_k\right) \bmod 1,$$

with $\vec{\Delta}_k$ being a uniform random in $[0;1)^n$, and the generating vectors $\vec{g}_N$ constructed separately for each N (up to $6{\cdot}10^{10}$) and $\alpha = 2$ (i.e. allowing for N^{-2} error scaling if the integrand is smooth enough). K is by default 32.

Monte Carlo vs RQMC

Integration time scaling for Monte Carlo (VEGAS) vs Randomized Quasi Monte Carlo (QMC).[1]



Vegas
RQMC+baker
RQMC+korobov

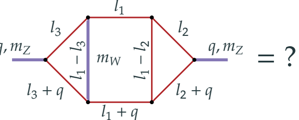
$N^{-0.5}$
$N^{-0.8}$
$N^{-1.6}$

Relative integration error: 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}, 10^{-7}

Integration time, s: 10^{1}, 10^{2}, 10^{3}

[1] pySECDEC v1.5.3 on NVidia A100 GPU.

Using pySecDec for a single Feynman integral



Generate the integration library:

```python
import pySecDec as psd
if __name__ == "__main__":
    psd.loop_package(
        name="dia1",
        loop_integral=
            psd.LoopIntegralFromPropagators(
                loop_momenta=["l1", "l2", "l3"],
                propagators=[
                    "(l1)**2",
                    "(l2)**2",
                    "(l3)**2",
                    "(l1 + q)**2",
                    "(l2 + q)**2",
                    "(l3 + q)**2",
                    "(l1 - l2)**2",
                    "(l1 - l3)**2 - mw2"
                ],
                powerlist=[1,1,1,1,1,1,1,1],
                replacement_rules=[
                    ("q*q", "mz2")
                ]),
        real_parameters=["mz2", "mw2"],
        requested_orders=[0],
        decomposition_method="geometric")
```

Inspect the generated code:

```
$ ls dia1/
dia1_data/          dia1_integral/  pylink/
src/                Makefile        Makefile.conf
README              dia1.hpp        integral_names.txt
integrate_dia1.cpp
```

Compile it for the CPU:

```
$ cd dia1 && make
```

or for both CPU and GPU (with CUDA):

```
$ export SECDEC_WITH_CUDA_FLAGS="-arch=sm_80" CXX=nvcc
$ cd dia1 && make
```

Run it:

```python
from pySecDec.integral_interface import IntegralLibrary
if __name__ == "__main__":
    lib = IntegralLibrary("dia1/dia1_pylink.so")
    lib.use_Qmc(verbosity=1)
    mz2 = 1.0
    mw2 = 0.78
    _, _, result = lib([mz2, mw2], epsrel=1e-7, verbose=True)
    print(result)
```

The result:

```
+ ((6.82645729748523067e+00,-1.60711801420445148e+01)
+/- (8.07205205949069617e-07,7.55815020343424309e-07))
+ O(eps)
```

Expected performance for 3-loop EW integrals

pySecDec[2] + QMC *integration times* for 3-loop self-energy integrals:[3]

Diagram \ Relative precision		10^{-3}	10^{-4}	10^{-5}	10^{-6}	10^{-7}	10^{-8}
Diagram 1 (m_Z, m_W, m_Z)	GPU	15s	20s	40s	200s	13m	50m
	CPU	10s	50s	400s	4000s	180m	1200m
Diagram 2 (m_Z, m_t, m_t, m_t, m_t, m_t, m_t, m_Z)	GPU	18s	19s	30s	20s	1.2m	2m
	CPU	5s	14s	60s	50s	12m	16m
Diagram 3 (m_Z, m_t, m_W, m_t, m_Z)	GPU	6s	11s	12s	30s	3m	24m
	CPU	5s	10s	50s	800s	60m	800m

[Same diagrams as in Dubovyk, Usovitsch, Grzanka '21]

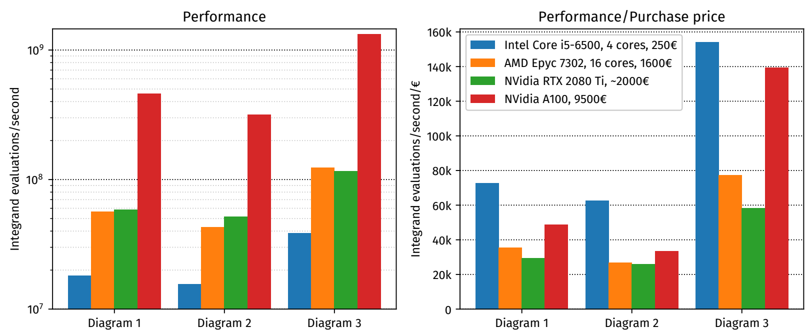
In short: *seconds to minutes per integral* to achieve practical precision.

[2] Version 1.5 + work-in-progress + AVX2.

[3] *GPU*: NVidia A100 40GB; *CPU*: AMD Epyc 7302 with 32 threads.

CPU vs GPU with pySecDec

pySecDec[4] integrand sampling speed on different devices:



In short:

- Top consumer-grade GPU (RTX 2080 Ti) ≈ server-grade CPU.
- Top server-grade GPU (A100) ≈ $10\times$ server-grade CPU.

[4] Version 1.5 + work-in-progress + AVX2.

On the choice of the master integrals

Integration time of similarly looking integrals:[5]

	top pole	t, s		top pole	t, s
m_Z m_W	ε^{-3}	27	m_Z m_W	ε^{-2}	57
m_Z m_W	ε^{-2}	1230	m_Z m_W	ε^{-2}	>9000

Takeaway: for best performance, test the integration speed and adjust the selection of the master integrals.

[5] To 10^{-3} precision, pySecDec 1.5.3, NVidia A100 GPU.

Adaptive sampling of amplitudes

Amplitude term	Naive sampling	Naive error	Better sampling	Better error
1 (sunset diagram)	10^6 samples	$1 \cdot 10^{-6}$	$\frac{1}{2} \cdot 10^6$ samples	$2 \cdot 10^{-6}$
10 (diamond diagram)	10^6 samples	$10 \cdot 10^{-6}$	$\frac{1}{2} \cdot 10^6$ samples	$20 \cdot 10^{-6}$
50 (double bubble diagram)	10^6 samples	$50 \cdot 10^{-6}$	$2 \cdot 10^6$ samples	$25 \cdot 10^{-6}$
Total:	$3 \cdot 10^6$	$51 \cdot 10^{-6}$	$3 \cdot 10^6$	$32 \cdot 10^{-6}$

[Example assumes integration error = $1/n$]

pySecDec now automatically optimizes the total integration time based on

- how fast each integral can be sampled,
- how well it converges,
- how large its coefficient is.

$\Rightarrow$ Automatic *speedup for amplitudes* (weighted sums of integrals).

$\Rightarrow$ Automatic speedup for single integrals too (sums of sectors).

$\Rightarrow$ Already used in 2-loop $gg \to ZH$ (2011.12325), $gg \to \gamma\gamma$ (1911.09314), and $H + \text{jet}$ (1802.00349).

Adaptive optimization: the problem formulation

Suppose we need to know A_j, some weighted sums of integrals I_i:

$$A_j \equiv \sum_i W_{ji} I_i.$$

Each I_i is a random variable with:

$$I_i \sim \mathcal{N}\left(\mathrm{mean}(I_i), \mathrm{var}(I_i)\right).$$

Assume that $\mathrm{var}(I_i)$ scales with the number of integrand evaluations n_i as

$$\mathrm{var}(I_i) = \frac{w_i}{n_i^\alpha}, \qquad \text{for some } \alpha.$$

Problem: choose n_i (as functions of W_{ji}, w_i, α, and τ_i) to minimize the total integration time T while achieving the requested total variance V_j:

$$T \equiv \sum_i \tau_i n_i \to min, \qquad \mathrm{var}\left(A_j\right) = \sum_i \left|W_{ji}\right|^2 \frac{w_i}{n_i^\alpha} = V_j \;\; \left(\forall j\right).$$

Adaptive optimization: the solution

Solution: via the Lagrange multiplier method,

$$L \equiv T + \sum_j \lambda_j \left(\mathrm{var}\left(A_j\right) - V_j\right), \qquad \text{and} \qquad \frac{\partial L}{\partial \left\{n_i, \lambda_j\right\}} = 0.$$

Solution if only one A_j is needed:

$$\lambda_j = \frac{1}{\alpha} \left(\frac{1}{V_j} \sum_k \left(\left|W_{jk}\right|^2 w_k \tau_k^\alpha \right)^{\frac{1}{\alpha+1}} \right)^{\frac{\alpha+1}{\alpha}}, \quad n_i = \left(\frac{\alpha w_i}{\tau_i} \lambda_j \left|W_{ji}\right|^2 \right)^{\frac{1}{\alpha+1}}.$$

Solution if there are many A_j: only approximate.
In practice pySecDec will:

1. Evaluate each I_i on $n_i = 10^4$ points, estimating w_i and τ_i.
2. Get the n_i values as above, but cap them at 20x the previous values.
3. Evaluate I_i using the updated n_i, update w_i and τ_i.
4. Check if $\mathrm{var}\left(A_j\right) \leq V_j$, repeat from 2 if needed.

Using pySecDec for amplitudes

Generate the integration library:

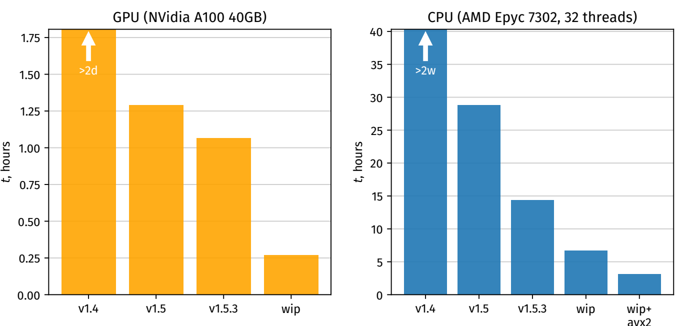
```
import pySecDec as psd
if __name__ == "__main__":
    # First term
    term1 = psd.LoopPackage(
        name="integral1",
        loop_integral=
            psd.LoopIntegralFromPropagators(
                ...
            ),
        real_parameters=[...])
    coeff1 = psd.Coefficient(
        numerators=['1 + 2*eps^3', ...],
        denominators=['-3 + 2*eps', '-1 + 2*eps', ...],
        parameters=[])
    # Second term
    term2 = ...
    coeff2 = ...
    ...
    # The amplitude
    psd.sum_package('amplitude',
        [term1, term2, ...],
        regulators=['eps'],
        requested_orders=[0],
        coefficients=[[coeff1, coeff2, ...]],
        real_parameters=[...]
    )
```

Compile an *run*: same as for a single integral.

$\Rightarrow$ Under the hood single integrals are implemented the same as sums.

Performance improvements by pySecDec version

Time to integrate [diagram: m_Z, m_W, m_Z] to 7 digits of precision with pySecDec + Qmc:



Speedup sources:

- *v1.5:* adaptive sampling, automatic contour deformation adjustment;
- *v1.5.3:* separation of real and complex variables in the integrand code;
- *wip:* simlification of the integrand code, vectorization on CPU (AVX2).

The latest release is fast; *the next release will be faster.*

Generated code: v1.5 vs wip

pySecDec *v1.5* & Qmc:

```
integrand_return_t sector_1_order_0_integrand(
        const real_t * const integration_variables,
        const real_t * const realp,
        const complex_t * const complexp,
        const real_t * const deformp,
        secdecutil::ResultInfo * const result_info) {
    real_t mz2 = realp[0], mw2 = realp[1];
    real_t x0 = integration_variables[0];
    [...]
    integrand_return_t tmp[49];
    tmp[1] = x2*x1;
    tmp[2] = tmp[1] + x1;
    tmp[3] = tmp[2]*mz2;
    tmp[4] = mz2*x2;
    [...]
    return tmp[0];
};
[...]
template<...> void compute(
        const U index0,
        const std::vector<U>& genvec,
        const std::vector<D>& shift,
        T* results,
        const U r_size_over_m,
        const U stride, const U latsize, const U nshifts,
        I& func) {
    for (U k = 0; k < nshifts; k++) {
        for (U i = index0; i < latsize; i += stride) {
            std::vector<D> x(func.dimension, 0);
            for (U j = 0; j < func.dimension; j++) {
                x[j] = modf(
                    mul_mod(i, genvec[j], latsize)/latsize
                        + shift[k*func.dimension+j]);
            }
            T point = func(x.data());
            results[k*r_size_over_m] += point;
} } }
```

pySecDec *work in progress*:

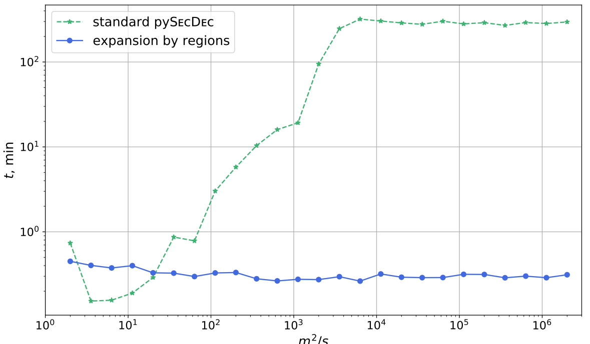
```
int integral__sector_1_order_0(
        result_t * restrict result,
        uint64_t latsize,
        uint64_t index1, uint64_t index2,
        const uint64_t * restrict genvec,
        const real_t * restrict shift,
        const real_t * restrict realp,
        const complex_t * restrict complexp,
        const real_t * restrict deformp) {
    const real_t mz2 = realp[0];
    const real_t mw2 = realp[1];
    [...]
    resultvec_t sum = {0, 0, 0, 0};
    int64_t latidx_x0 = mulmod(genvec[0], index, latsize);
    [...]
    for (uint64_t i = index1; i < index2; i += 4) {
        int64_t li_x0_0 = latidx_x0;
        latidx_x0 = modonce(latidx_x0 + genvec[0], latsize);
        [...]
        realvec_t x0 = {{ li_x0_0, li_x0_1, li_x0_2, li_x0_3 }};
        x0 = modonce(x0*(1.0/latsize) + shift[0], 1);
        [...]
        realvec_t jac = korobov3_w(x0) * korobov3_w(x1) * [...];
        [...]
        x0 = korobov3_f(x0);
        [...]
        auto tmp1_1 = x1*mw2;
        auto tmp1_2 = 2*tmp1_1;
        auto tmp1_3 = tmp1_2 + mw2;
        auto tmp1_4 = tmp1_3*x3;
        auto tmp1_5 = tmp1_4 + mw2;
        [...]
        sum = sum + jac*(tmp3_356);
    }
    *result = componentsum(sum);
    return 0;
}
```

Asymptotic expansion: the motivation

$$I \equiv \text{[diagram]} = \int \mathrm{d}x_1 \cdots \mathrm{d}x_6 \, \frac{\mathcal{U}^{3\varepsilon}(\vec{x})}{\mathcal{F}^{2+2\varepsilon}(\vec{x}, s, m^2)} = ?$$

Problem: when $s/m^2 \ll 1$ numerical integration converges poorly.

E.g. the time to integrate I to 10^{-3} accuracy:[6]



- This is a general problem when scale ratios are not ≈ 1.

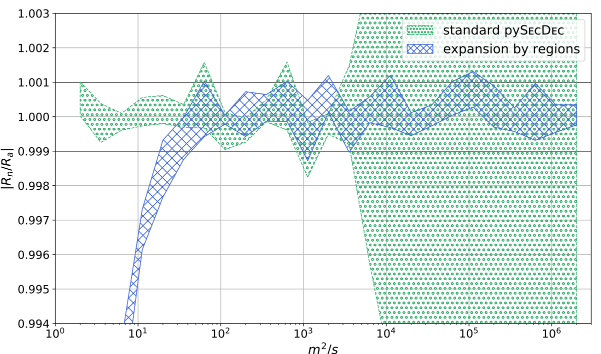
[6] pySecDec v1.5, NVidia A100 GPU.

Asymptotic expansion

Solution: take out the extreme ratios (s/m^2) from the integrand via asymptotic expansion:

$$I = (\cdots + \cdots)\left(\frac{s}{m^2}\right)^{-1} + (\cdots + \cdots + \cdots + \cdots)\left(\frac{s}{m^2}\right)^{0} + \mathcal{O}\left(\frac{s}{m^2}\right)$$

Compare the exact result and the asymptotic expansion:



When $\frac{s}{m^2} < 10^{-2}$ both match up to 0.1%.

Asymptotic expansion: the method

Method of *expansion by regions*: [Beneke, Smirnov '98]

- Split the integration space into *regions*, ordering the variables by the smallness parameter (s/m^2) in each.
  - E.g. $\{x_1, \ldots, x_6\} \sim \left(\frac{s}{m^2}\right)^{\{-1,-1,-1,-1,-1,0\}}$, $\sim \left(\frac{s}{m^2}\right)^{\{0,0,-1,-1,-1,0\}}$, etc.
  - The required regions can be found geometrically. [Jantzen '11]
- Taylor-expand the integrand in each region (in s/m^2).
- Integrate and sum the regions.
  - Can integrate over the whole integration space in each region.

Implementations:

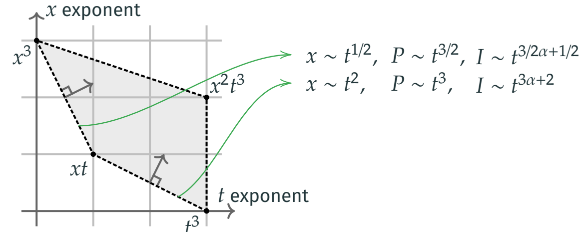
- ASY2.M (part of FIESTA); [Jantzen, Smirnov, Smirnov '12]
- pySecDec v1.5 (via `pySecDec.loop_regions`). [Heinrich et al '21]

Asymptotic expansion: finding the regions

Consider an integral depending on *small parameter t* like

$$I = \int_0^1 P^\alpha(x)\,\mathrm{d}x, \qquad P = a\,x^3 + b\,t\,x + c\,t^3 + d\,t^3\,x^2.$$

Plot the *Newton polytope* of P:



From the facets of the polytope we find *two regions*:

1. $x \sim t^{1/2}$, and P can be expanded as $\left(ax^3 + btx\right)\left(1 + \frac{ct^3+dt^3x^2}{ax^3+btx}\right)$;
2. $x \sim t^2$, and P can be expanded as $\left(btx + ct^3\right)\left(1 + \frac{ax^3+dt^3x^2}{btx+ct^3}\right)$.

Note: only valid if $a, b, c > 0$.

Using pySecDec with asymptotic expansion

Generate the integration library:

```
import pySecDec as psd
if __name__ == "__main__":
    # find the regions and expand the integral up to O(s)
    terms = psd.loop_regions(
        name="triangle2L",
        loop_integral=
            psd.LoopIntegralFromPropagators(
                loop_momenta=["l1", "l2"],
                external_momenta=["p1", "p2"],
                propagators=[
                    "(l1 + p1)**2",
                    "(l1 - p2)**2",
                    "(l2 + p1)**2",
                    "(l2 - p2)**2",
                    "l2**2",
                    "(l1 - l2)**2 - msq"
                ],
                replacement_rules=[
                    ("p1*p1", 0),
                    ("p2*p2", 0),
                    ("p1*p2", "s/2")
                ]),
        smallness_parameter="s",
        decomposition_method="geometric",
        expansion_by_regions_order=0)
    # generate the library
    psd.sum_package("triangle2L_by_regions",
        terms,
        regulators=["eps"],
        requested_orders=[0],
        real_parameters=["s", "msq"])
```

Compile an *run*: same as for a single integral.

⇒ Adaptive amplitude optimization is enabled automatically.

Summary

pySecDec provides:

- Numerical evaluation of *massive multi-loop integrals*.
  - many masses of similar magnitude (m_t, m_H, m_Z, m_W) are no problem;
  - higher loop count is no problem;
  - 3-loop massive integrals at 6 digits in minutes.
- Results when other methods fail.

The latest release (v1.5) includes: [Heinrich '21]

- Optimized evaluation of *amplitudes* (weighted sums of integrals).
  - Proven in multiple 2-loop calculations.
  - Automatically applicable to single integrals too.
- *Asymptotic expansion* of integrals for e.g. high-energy regions.
- Usability improvements.

Future releases will bring:

- Even better performance.